UCRL- 100195
PREPRINT

A SLAP for the Masses

Mark K. Seager

This paper was prepared for presentation at
"Methods & Algorithms for
PDE's on Advanced Processors"
Austin, TX
October 17-18, 1988

December 12, 1988

Lawrence
Livermore
National
Laboratory

A SLAP for the Masses †

Mark K. Seager
User Systems Division
Lawrence Livermore National Laboratory

December 8, 1988

Abstract

A Sparse Linear Algebra Package (SLAP), written in FORTRAN77, for the iterative solution of large sparse symmetric and non-symmetric linear systems is presented. SLAP Version 2.0 consists of three levels of routines: "high level", "core" and "utility." The "core" routines implement the following preconditioned iterative methods: *iterative refinement, conjugate gradient, conjugate gradient on the normal equations, bi-conjugate gradient, bi-conjugate gradient squared, orthomin* and *generalized minimum residual.* All of these methods do not require the data structure of the matrix being solved nor of the preconditioning matrix, but do require the "user" to supply a matrix vector product and preconditioning routines. The "high level" routines assume one of two specific data structures and provide the required "user routines." The preconditioners supported are *diagonal scaling* and *incomplete factorization.* One of the SLAP data structures allows for the vectorization of the matrix multiply and the backsolve of the incomplete factorization operations on machines with hardware gather/scatter capabilities. We present results for SLAP on the Cray Y/MP and Alliant FX/8 machines.

†This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, Department of Energy, by Lawrence Livermore National Laboratory under contract W-7405-ENG-48.

1 Introduction

The Sparse Linear Algebra Package SLAP was conceived in 1985 in discussions with members of the Computing and Mathematics Research Division at Lawrence Livermore National Laboratory [5]. It had three basic design criteria:

1. Easy of use for non-experts.
2. Ability to switch between methods with a minimum of effort.
3. Utility routines to various matrix manipulations and viewing.

The package was then implemented mostly by Anne Greenbaum as a set of iterative and direct methods for use on the Cray X/MP supercomputers. The original data structure (SLAP Triad format, descried in Section 4) has the advantage that it is trivial to use, but has the draw back that none of the matrix vector operations vectorize. This version of SLAP also contained translation routines to various banded and sparse data structures so that one could use sparse direct methods as well. The user interface was simplified in that no workspace was required from the user: it was obtained from the system using Cray Time Sharing System (CTSS) specific GETSPACE, RELSPACE memory allocation/deallocation routines and Cray FORTRAN compiler (CFT) pointer statements. In addition a matrix viewer routine was set up to utilize Livermore specific interactive graphics output devices (TMDS). We got comments back from the users that they liked this type of user interface very much, but it was slow (due to the non-vectorizability of the matrix vector operations) and was not portable (due to the memory allocation scheme and graphics output devices).

In 1987 a new set of design goals were established:

1. Portability and conformance to SLATEC [4] standards.
2. Ease of use for the non-expert.
3. Vectorization of matrix vector operations.
4. Ability to switch between iterative methods with a minimum effort.
5. Utility routines for matrix manipulation and viewing.

SLAP was rewritten to conform to these new design constraints by Anne Greenbaum and the Author. First, the internal memory allocation (via pointers) was done away with and the necessary workspace obtained from the "user" via the subroutine calling sequence. This, of course, complicates the use of the package (making sure one has given the right amount of workspace for various routines is quite error prone, especially for non-experts), but until the FORTRAN standard is changed to handle some type of memory allocation this is the best anyone can do while remaining portable. Additional modifications to the package included a consistent naming and internal documentation (FORTRAN comments) conventions based on the **SLATEC** standard. Finally, two new iterative methods were added for non-symmetric systems: *generalized minimum residual* (GMRES) [9] and *bi-conjugate gradient squared* (BCGS) [10] and the direct methods were dropped.

2 Data Structures

The core routines of **SLAP** are written in such a way so that the data structure of the matrix and its associated preconditioner are only referenced referenced in the "user supplied" routines `MatVec` and `MSolve`. Hence, the core routines are completely independent of the "user supplied" data structure. This allows the package great flexibility in that any method can be plugged into a production code environment (i.e., conform to a highly structured data environment) if the user is willing to go to the work of writing preconditioning and matrix vector multiply routines.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & 0 & 0 & a_{1,5} \\ a_{2,1} & a_{2,2} & 0 & 0 & 0 \\ 0 & 0 & a_{3,3} & 0 & a_{3,5} \\ 0 & 0 & 0 & a_{4,4} & 0 \\ a_{5,1} & 0 & a_{5,3} & 0 & a_{5,5} \end{pmatrix} \Longrightarrow \begin{array}{cccc} \textit{Index} & \textit{Element} & \textit{Row} & \textit{Column} \\ 1 & a_{5,1} & 5 & 1 \\ 2 & a_{1,2} & 1 & 2 \\ 3 & a_{1,1} & 1 & 1 \\ 4 & a_{3,3} & 3 & 3 \\ 5 & a_{1,5} & 1 & 5 \\ 6 & a_{5,3} & 5 & 3 \\ 7 & a_{5,5} & 5 & 5 \\ 8 & a_{2,2} & 2 & 2 \\ 9 & a_{3,5} & 3 & 5 \\ 10 & a_{4,4} & 4 & 4 \\ 11 & a_{2,1} & 2 & 1 \end{array} \tag{2.1}$$

Figure 2.1: SLAP Triad Matrix Storage Format.

On the other hand, some code development environments do not so constrain programmers and they may choose utilize a data structure supported by SLAP. In this situation the "high level" SLAP routines can be used with a choice of two matrix data structures. The simplest data structure is that know as SLAP Triad format (also known as "coordinate format"). See Figure 2.1. With this structure only non-zeros of the matrix are stored, in any order, in one `real` array `A`. If the matrix is symmetric only the lower triangle (including the diagonal) need be stored. Suppose `nelt` is the number of elements stored in `A`. Then two additional `integer` arrays of length `nelt`, `IA` and `JA` are needed in the SLAP Triad format to hold the row and column indices of the matrix elements, respectively. This format is not the most storage efficient, but it is trivial to set up. Unfortunately, the other drawback of this matrix storage mode is that the matrix vector multiply and incomplete factorization backsolves (preconditioning step) do not vectorize. So if the user decides, for convience, to use this matrix data structure the package transforms it automatically to the SLAP Column format.

$$
\begin{pmatrix}
a_{1,1} & a_{1,2} & 0 & 0 & a_{1,5} \\
a_{2,1} & a_{2,2} & 0 & 0 & 0 \\
0 & 0 & a_{3,3} & 0 & a_{3,5} \\
0 & 0 & 0 & a_{4,4} & 0 \\
a_{5,1} & 0 & a_{5,3} & 0 & a_{5,5}
\end{pmatrix}
\Longrightarrow
\begin{array}{cccc}
Index & Element & Row & ColumnPointer \\
1 & a_{1,1} & 1 & 1 \\
2 & a_{2,1} & 2 & 4 \\
3 & a_{5,1} & 5 & 6 \\
4 & a_{2,2} & 2 & 8 \\
5 & a_{1,2} & 1 & 9 \\
6 & a_{3,3} & 3 & 12 \\
7 & a_{5,3} & 5 & \\
8 & a_{4,4} & 2 & \\
9 & a_{5,5} & 5 & \\
10 & a_{1,5} & 1 & \\
11 & a_{3,5} & 3 &
\end{array}
\tag{2.2}
$$

Figure 2.2: SLAP Column Matrix Storage Format.

The SLAP Column format is similar to the SLAP Triad format in that the non-zeros of the matrix are stored in a `real` array `A` and that the corresponding row indices are stored in an `integer` array `IA`. On the other hand, this data structure differs from the SLAP Triad format in that the matrix elements must be stored in a very specific order and the `JA` array a has completely different interpretation. In the SLAP Column format all the non-zeros of the matrix elements must be stored by column starting with column 1 and ending with column `n`, where `n` is the number of unknowns in the problem. Secondly, the diagonal element must be the first entry stored for each column. The `JA` array is used in this context as offsets into the `A` and `IA` arrays for the the beginning of the compressed columns. In order to simplify coding, the last element of `JA`, `JA(n+1)`, points to the beginning of column `N+1`, which is imaginary. See the example in Figure 2.2. In other words, `A(JA(k))` and `IA(JA(k))` are the first stored matrix element and row index for the k^{th} column and `JA(n+1)` points just past the last stored matrix element and index. An example of how to use this data structure a matrix vector multiply routine is given in Figure 2.3. It is assumed that the matrix is symmetric and only the lower triangle and the diagonal are stored when `isym = 1`.

```
      subroutine MatVec( n, x, y, nelt, ia, ja, a, isym )
      integer n, nelt, ia(nelt), ja(*), isym
      real    x(n), y(n), a(nelt)
c
c        Compute y = A*x.
c
      do 10 i = 1, n
         y(i) = 0.0
 10   continue
      do 30 icol = 1, n
         ibgn = ja(icol)
         iend = ja(icol+1)-1
         do 20 i = ibgn, iend
            y(ia(i)) = y(ia(i)) + a(i)*x(icol)
 20      continue
 30   continue
      if( isym.eq.1 ) then
c
c        The matrix is symmetric.  Lower triangle is stored.
c        Multiply by the transpose, ignoring the diagonal.
c
         do 50 irow = 1, n
            jbgn = ja(irow)+1
            jend = ja(irow+1)-1
            if( jbgn.gt.jend ) GoTo 50
            do 40 j = jbgn, jend
               y(irow) = y(irow) + a(j)*x(ia(j))
 40         continue
 50      continue
      endif
      return
      end
```

Figure 2.3: Matrix Vector Multiply with SLAP Triad Format.

Note that the inner loops, `do 20 i = ibgn, iend` and `do 40 j = ibgn, iend`, should vectorize on machines with hardware gather/scatter capabilities.

3 Preconditioners

Two preconditioners are supplied with the **SLAP** package. They both assume the **SLAP** Column format (although the "user level" routines will detect **SLAP** Triad format and automatically transform to the **SLAP** Column format). The first preconditioner supplied is the symmetric *diagonal scaling* (DS). Suppose we are solving the linear system:

$$Ax = b, \tag{3.3}$$

For diagonal scaling this system is transformed to:

$$(D^{-\frac{1}{2}}AD^{-\frac{1}{2}})(D^{\frac{1}{2}}x) = D^{-\frac{1}{2}}b, \tag{3.4}$$

where $D = Diag(A)$ is the diagonal of A. This corresponds to setting the preconditioning matrix $M = D$ and is the simplest of all preconditioners. It only requires that the diagonal be positive. The benefits of this preconditioner are: 1) it reduces the iteration count on most "real world" problems; 2) it vectorizes very nicely.

The other preconditioner used in **SLAP** is *modified incomplete LU factorization* (LU) (*modified incomplete Cholesky factorization* (IC) if A is symmetric). Here, one chooses the preconditioning matrix to be the LU (or LL^t if A is symmetric) factorization of the original system without allowing any additional non-zeros (fill-in) to be created. It has been shown that if A is an M-matrix or an H-matrix [8] then the incomplete Cholesky factorization exists. In general the method can break down and **SLAP** follows the strategy of Kershaw [7]: setting non-positive diagonal elements of the incomplete Cholesky factorization to unity. For most problems that are not too ill-conditioned this preconditioning reduces the iteration count substantially. On the other hand, the cost of computing the incomplete factorization is non-trivial and the vectorization of the backsolves (which must be done on each iteration) is necessarily done with short vectors and indirect addressing. Hence, *modified incomplete factorization* is not a clear winner over *diagonal scaling* for all linear systems solved on vector supercomputers.

4 Iterative Methods

The SLAP package contains preconditioned iterative methods for both symmetric and non-symmetric linear systems. Section 3 discusses the preconditioners supplied with the package. For symmetric systems one can choose from *iterative refinement* (IR), *Jacobi* (JAC), *Gauss-Seidel* (GS) and *conjugate gradient* (CG) iterative methods. The *iterative refinement*, *Jacobi* and *Gauss-Seidel* are included just for comparison purposes and are not normally used for "real problems." The computational *Conjugate Gradient* algorithm used in this package is well known and can be found in [1]. See Table 4.1 for a tabulation of storage requirements and work estimates for these methods. In Table 4.1 and Table 4.2 the units of storage are: n (the number of unknowns), nl (the number of nonzeros stored in the lower triangle of the matrix, including the diagonal), nu (the number of nonzeros stored in the upper triangle of the matrix, including the diagonal), $nelt = nu + nl - n$ (the total number of nonzeros stored), MatVec (the number of operations required to perform a matrix vector multiply with the "user supplied" routine) and MSolve (the number of operations required to perform a preconditioning step with the "user supplied" routine). For the routines where SLAP provides the `MatVec` and `MSolve` the work estimates are given in terms of n, nl, nu and $nelt$.

Basic Methods				
Subroutine	Method	ISTORE	RSTORE	Work per Iteration
SIR	IR	0	3*n	2*n+MatVec+MSolve
SSJAC	JAC	10	4*n	2*n+2*nl
SSGS	GS	nl+n+11	nl+3*n	4*nl
SSILUR	ILU IR	nelt+3*n+11	nelt+3*n	4*n+6*nelt
SCG	CG	0	3*n	10*n+MatVec+MSolve
SSDCG	DS CG	10	4*n	10*n+2*nl
SSICCG	IC CG	nl+2*n+11	nl+5*n	8*n+4*nl

Table 4.1: Storage and Work for Basic Iterative Methods.

The largest number of methods in SLAP are for non-symmetric systems. If the A matrix is positive definite ($(Ax, x) > 0$ for all non-zero N-vectors x), then one can apply *conjugate gradient on the normal equations* (CGN) for Equation 3.3, viz.,

$$A^T Ax = A^T b. \tag{4.5}$$

Rather than form $A^T A$ directly, which may not have a very sparse structure, we rewrite the *conjugate* *gradient* algorithm in terms of both A and A^T. This then requires two matrix vector multiplies per iteration (one of which by the transpose of the matrix). Hence, for CGN (and most of the other methods for non-symmetric systems) the user must also supply a `MTtVec` routine (for the matrix transpose times a vector operation) as well as a `MatVec` routine. CGN has not been used in the past due to the fact that using the normal equations squares the condition number of the resulting iteration matrix. This causes this method to converge very slowly. Recently, this method has gained some popularity due to the fact that when a good preconditioner to A^TA can be found it counteracts the effect of squaring the condition number, thereby making the method competitive.

Bi-conjugate gradient (BCG) was proposed by Fletcher [3] for indefinite systems. It is similar to *conjugate gradient* in that it requires A^T as well as A, but it also requires the transpose of the preconditioner M^T. Hence the user interface to *bi-conjugate gradient* requires four routines (`MatVec`, `MTtVec`, `MSolve` and `MTSolve`). The theoretical properties of *bi-conjugate gradient* are not very pleasing. In particular, if A is non-symmetric (the only case when one would BCG) then *bi-conjugate gradient* is not guaranteed to reduce any quadratic functional (as in the case of *conjugate gradient*). In practice this does not seem to be a problem and if one has a good preconditioner *bi-conjugate gradient* is an effective method.

When the *bi-conjugate gradient* method is converging another algorithm developed by Sonneveld [10] converges twice as fast: *bi-conjugate gradient squared* (BCGS). In addition to this quality *bi-conjugate gradient squared* also does not require A^T nor M^T and hence is much easier to use. It has been observed by the author and others that on problems where BCG is diverging BCGS will also diverge and twice as fast! In addition, on problems where *bi-conjugate gradient* seems to stagnate (i.e., not reduce the error for a large number of iterations) before converging, *bi-conjugate gradient squared* will more likely than not end up diverging. Hence, it is recommended that one use *bi-conjugate gradient* on the problem one wants to solve and if the convergence is acceptable then apply *bi-conjugate gradient squared* (with the same preconditioner). Due the the modular nature of SLAP, it is very easy to switch between methods in this manner.

Another extension of the *conjugate gradient* method to non-symmetric systems is *orthomin* (OMIN(k)) by Vinsome [6]. For this algorithm one chooses the next search direction as a linear combination of the previous residual and k previous search directions so that the new search direction is A-orthogonal to the previous k search directions. If one then chooses to minimize the norm of the residual along this search direction then the orthomin algorithm is obtained. When k is chosen to be the iteration count (i.e. one A-orthogonalizes against all the previous search directions) then *orthomin* is guaranteed not to break down.

A Lanczos type extension of *conjugate gradient* for general non-symmetric systems can be found in the *generalized minimum residual* (GMRES) method of Saad and Schultz [9]. It generates an orthonormal basis from the Krylov subspace:

$$K(l) = span\{r_0, Ar_0, A^2r_0, \ldots, A^{l-1}r_0\}, \tag{4.6}$$

where $r_0 = b - Ax_0$ is the initial residual, via the Arnoldi process. GMRES finds the approximate solution x_l in the affine subspace $x_0 + K(l)$ which has the minimal residual norm. This n-dimensional least squares problem can be reduced to a smaller l-dimensional least squares problem. GMRES is guaranteed to converge to the true solution in $l \leq n$ iterations for any non-singular matrix A. Usually however, a maximum value of l, denoted by k, is dictated by storage considerations to be very much smaller than n. If the stopping test is not met within k iterations, the iteration can be restarted by setting $x_0 = x_k$ and applying the GMRES algorithm again, and so on. This algorithm is denoted by GMRES(k), and is guaranteed to converge as long as A is positive definite. A default value of $k = 10$ is used in the SLAP implementation of GMRES(k), but this value can be optionally set by the user. In building up the Krylov subspace $K(l)$ it is possible to estimate the error with $\frac{\|r_l\|}{\|b\|}$ without having to compute either the intermediate solution x_l or residual r_l. If one can utilize this "natural" GMRES(k) stopping test, then the algorithm is very efficient.

See Table 4.2 for detailed storage requirement and work estimate for the non-symmetric methods as implemented in SLAP. As in Table 4.1, the storage requirements and work estimates are given in terms of n, k, MatVec and MSolve for the "core" methods and in terms of n, k, nl, nu and $nelt$ for the "high level" methods. The work description of GMRES(k) is not entirely accurate since the cost of restarting is not accounted for.

SLAP 2.0 iterative methods, matrix vector and preconditioner calculation routines follow a naming

Non-Symmetric Methods				
Subroutine	Method	ISTORE	RSTORE	Work per Iteration
SCGN	CGN	0	3*n	13*n+2*MatVec+2*MSolve
SSDCGN	DS CGN	10	4*n	15*n+4*nelt
SSLUCN	LU CGN	nl+n+11	nl+3*n	17*n+12*n
SBCG	BCG	0	7*n	14*n+2*MatVec+2*MSolve
SSDBCG	DS BCG	10	8*n	16*n+4*nelt
SSLUBC	LU BCG	nl+nu+4*n+12	nl+nu+8*n	18*n+12*nelt
SCGS	BCGS	0	7*n	17*n+2*MatVec+2*MSolve
SSDCGS	DS BCGS	10	8*n	19*n+4*nelt
SSLUCS	LU BCGS	nl+nu+4*n+12	nl+nu+8*n	21*n+12*nelt
SOMN	OMIN(k)	0	n*(6+3*k)+k	(11+8*k)*n+MatVec+MSolve
SSDOMN	DS OMIN(k)	10	n*(7+3*k)+k	(12+8*k)*n+2*nelt
SSLUOM	LU OMIN(k)	nl+nu+4*n+12	nl+nu+n*(7+3*k)+k	(13+8*k)*n+6*nelt
SGMRES	GMRES(k)	20	n*(k+6)+k*(k+3)+1	3*k*n+MatVec+MSolve
SSDGMR	DS GMRES(k)	30	n*(k+7)+k*(k+3)+1	(1+3*k)*n+nelt
SSLUGM	LU GMRES(k)	nl+nu+4*n+32	nl+nu+n*(k+7)+k*(k+3)	(2+3*k)*n+2*nelt

Table 4.2: Storage and Work for Basic Iterative Methods.

convention which, when understood, allows one to determine the iterative method, preconditioner and data structure(s) used in the routine. The subroutine naming convention takes the following form:

`P[F][M]D,`

where `P` stands for the precision (or data type) of the routine and is required in all names, the format code `F` denotes whether or not the routine requires the SLAP Triad or Column format (it requires the Column format if the second letter of the name is `S` otherwise it is matrix storage format independent), the optional `M` stands for the type of preconditioner used (only appears in drivers for "core" routines) and `D` is some number of letters describing the method or purpose of the routine. In this incarnation of SLAP only single precision data types are supported (no double precision or complex data type routines have been written). Hence, all routines start with the letter `S`. The brackets around `S` and `M` designate that these fields are optional.

The possibilities for the preconditioning, `M`, field are: D (diagonal scaling); IC (modified incomplete Cholesky factorization); ILU or LU (modified incomplete *LU* factorization). The description field, `D`, possibilities are: IR or R (iterative refinement); JAC (Jacobi); GS (Gauss Seidel); BCG or BC (biconjugate gradient); CG (conjugate gradient); CGS or CS (biconjugate gradient squared); GMRES, GMR or GM (generalized minimum residual); OMN or OM (orthomin); DS (diagonal scaling preconditioner setup); D2S (diagonal scaling for normal equations preconditioner setup); 2LT (lower triangle preconditioning setup); ICS (incomplete Cholesky decomposition preconditioning setup); ILUS (incomplete

LU decomposition setup); MV (matrix vector multiply); MTV (matrix transpose vector multiply); DI (SLAP solve for diagonal scaling); LI and LI2 (SLAP solve for lower triangle preconditioning); LLTI and LLTI2 (incomplete Cholesky SLAP solve); LUI and LUI2 (incomplete LU SLAP solve); LUTI and LUI4 (incomplete $(LU)^T$ SLAP solve); MMTI and MMI2 (SLAP solve for incomplete factorization preconditioning of the normal equations).

5 Utility Routines

The SLAP 2.0 package contains routines for manipulating data structures and for doing various matrix I/O. A short list of the routines and their purpose follows.

SBHIN Single precision routine that reads in a sparse matrix in the Boeing/Harwell format.

QS2I2R Sorts, using the quicksort algorithm, into ascending order an integer array carrying along one integer array and one real array. Can be used as the first step in transforming from the SLAP Triad format to the SLAP Column format.

SS2Y SLAP Triad format to SLAP Column format converter.

SCPPLT Printer plot of the SLAP Column format. For large matrices, only the first 132 rows and columns are displayed.

STOUT Prints out a matrix, right hand side and solution (or any combination which includes the matrix) in the SLAP Triad format.

STIN Reads in a matrix, right hand side and solution (or any combination which includes the matrix) in the SLAP Triad format.

6 Results

Four of the five problems (SHERMAN1, SHERMAN2, STEAM2 and JPWH991) used in these test came from the Harwell-Boeing sparse matrix collection [2]. SHERMAN1 is symmetric and arises from a three dimensional black oil with shale barriers oil reservoir simulator on a $10 \times 10 \times 10$ grid with one equation at each grid cell. SHERMAN2 is non-symmetric and also arises from three dimensional reservoir simulation. It is a problem that also involves simulation of steam injection into wells. The grid for this problem is 6 × 6 × 5 with 5 equations at each grid point. JPWH991 is non-symmetric and arises from circuit physics modeling. STEAM2 is non-symmetric and like SHERMAN1 arises from a three dimensional oil reservoir simulator. This problem is modeling enhanced oil recovery techniques by steam injection via a 5 × 5 × 5 grid with 4 variables at each grid cell. The only other problem with results listed below is the NASA1824 problem from H.D.Simon. This is a symmetric three dimensional structures problem. All these problems are considered difficult for both iterative and direct methods. For iterative methods, the preconditioner is of vital importance. For all the results given below, only the preconditioners supplied with the package (diagonal scaling and incomplete factorization) were used. No effort was made to tailor a preconditioner to any problem. This then can be considered the worse case for the iterative methods. Any production code developer would spend some amount of time adapting a preconditioner specialized to the problem being solved. In particular, for the three dimensional problems one would surely consider utilizing a preconditioner based on solving planes of two dimensional problems or some block factorization scheme.

Problem Descriptions					
Problem	Size	NELT	SymNrm	% NZ	RCond
NASA1824	1,824	20,516	0.000	1.23	4.77E-7
SHERMAN1	1,000	3,750	0.000	0.38	2.17E-4
JPWH991	991	6,027	0.092	0.61	4.03E-3
SHERMAN2	1,080	23,094	0.995	1.98	1.68E-12
STEAM2	600	13,760	0.002	3.82	3.53E-7

Table 6.3: Vital Statistics for Sample Problems.

All the above problems were solved with both direct methods (SGECO/SGESLand MA28) and iterative methods in the form of SLAP version 1.0 (the original Triad data structure, etc.) and version 2.0 (new column data structure, etc.). Table 6.3 gives the basic statistics for the test problems in terms of the linear system size (Size), number of non-zeros stored (NELT), the $SymNrm = \frac{\|A-A^t\|}{\|A\|}$ and percent nonzero $\%NZ = \frac{NELT}{Size^2}$ and the estimate of the condition number of the matrix $Rcond = \frac{1.0}{Condition(A)}$.

The direct method SGECO was chosen in order to obtain the condition number estimate as well as the factorization. The FORTRAN LINPACK implementation was used with hand coded level 1 BLAS.

For the MA28 runs the pivoting flag $u = 1.0$ (which implies partial pivoting for numerical stability) was chosen so that the method was as robust as possible.

The problems were solved on the Cray Y/MP832 (SN1002) at the NASA Ames NAS facility and the Alliant FX/8 (medusa.llnl.gov) at the Lawrence Livermore National Laboratory utilizing only one processor. Both machines have hardware support for vector gather/scatter operations. In the tables below the Cray Y/MP (single precision, 64 bit) results are given on the left hand side and the Alliant FX/8 (double precision, 64 bit) on the right. The *Time* results are in CPU Seconds and the *ITER* column is the number of iterations taken to solve the problems via the iterative methods to a tolerance of 10^{-6}. The *Int* and *Real* columns give the amount of integer and real workspace (respectively) required by the various methods. These numbers include the matrix, preconditioners and workspace. The solution and right hand side are not include in these statistics (since all methods must store these in the same fashion). For these test the number of past vectors stored for the Orthomin method was varied between eleven and one. For the GMRES method this parameter was varied between eleven and five.

Results for NASA1824						
	Cray Y/MP832		Alliant FX/8		Storage	
Method	ITER	Time	ITER	Time	Int	Real
SGECO		48.79		548.79	1,824	3,328,800
MA28		98.89		1,069.03	265,237	193,239
SSDCG	1,387	9.36	1,383	238.23	22,351	29,636
sdcg	1,386	18.49	1,382	760.47		
SSICCG	-5		-5		65,208	70,668
iccg	264	7.53	260	257.11		
SSDBCG	1,407	20.75	1,399	454.14	22,351	35,108
dsbcg	1,410	39.79	1,402	1,520.42		
SSLUBC	280	8.93	269	181.36	70,681	76,140
ilubcg	281	17.14	269	531.01		
SSDCGS	1,463	21.48	1,232	1,240.31	22,351	35,108
SSLUCS	387	11.32	311	218.22	70,681	76,140

Table 6.4: Time and Storage Results for NASA1824.

Table 6.4 gives the results for the symmetric system NASA1824. This problem is ill-posed enough so that the conjugate gradient method applied to the normal equations is not effective (i.e., did not converge) with diagonal scaling and incomplete factorization as preconditioners. Also, the Orthomin and GMRES methods did not converge for this problem. It is interesting to note that the incomplete Cholesky factorization for NASA1824 broke down and had to be modified [7]. The modification then produced a preconditioned system ($M^{-1}A$) that was not positive definite. This breakdown was noticed by `SSICCG` and the iteration was terminated (and hence the $ITER$ column shows the error return code of -5), but the **SLAP** version 1.0 routine `iccg` did not and was in fact able to compute a solution (not guaranteed by the mathematical theory). It is clear from these results that the diagonal scaling preconditioner is not competitive for this problem. The best method (`SSLUBC`), discounting `iccg` whose convergence is a matter of luck, was 5.46 times faster than the direct solve and 11.01 times faster than the sparse method on the Cray Y/MP (similar improvements are observed on the Alliant FX/8). The superiority of `SSLUBC` solution technique is also displayed in the much smaller amount of storage required.

Results for SHERMAN1						
	Cray Y/MP832		Alliant FX/8		Storage	
Method	ITER	Time	ITER	Time	Int	Real
SGECO		4.26		310.42	1,000	1,001,000
MA28		0.59		14.41	26,124	18,752
SSILUR	514	1.88	472	44.30	11,513	10,501
ilur	514	1.51	472	50.15		
SSDCG	232	0.22	230	11.13	4,761	8,751
sdcg	232	0.25	230	14.77		
SSICCG	39	0.16	40	4.37	8,137	11,126
iccg	39	0.13	40	4.75		
SSLUCN	225	2.06	222	39.82	11,513	14,501
ilucgn	225	1.36	222	45.13		
SSDBCG	232	0.87	230	18.92	4,761	11,751
dsbcg	232	0.48	230	24.67		
SSLUBC	39	0.37	40	7.59	11,513	14,501
ilubcg	39	0.25	40	8.58		
SSDCGS	199	0.38	186	15.07	4,761	11,751
SSLUCS	29	0.22	30	5.84	11,513	14,501
SSDOMN(7)	506	0.56	506	37.63	4,761	31,758
dsomn(7)	506	0.73	506	47.62		
SSLUOM(3)	61	0.24	58	6.67	11,513	22,504
iluomn(3)	61	0.22	58	7.47		
SSDGMR(11)	799	0.84	769	56.16	4,781	21,906
SSLUGM(11)	55	0.23	48	6.25	11,533	24,656

Table 6.5: Time and Storage for SHERMAN1.

The SHERMAN1 problem turned out to be negative definite and this fact was detected by the **SLAP** version 2.0 methods. The **SLAP** version 1.0 methods had various difficulties ranging from operand range error to simply not converging So the problem was recast, yielding the results in Table 6.5. For this problem the real winner was `SSICCG`. It was 27 times faster than `SGECO` and 3.41 times faster than `MA28` on the Cray Y/MP. Because this method is so sparse (about 3.8 nonzeros per column) implying very short vectors (average length of 3.8) for the column oriented approach of SLAP version 2.0. In this situation the preconditioning, matrix multiply and the incomplete factorization algorithms are slower than the Triad format scalar algorithms on the Cray Y/MP. Similar results were observed on the Cray X/MP416 (not presented here). Here is the first qualitative difference in the results obtained on the Cray computers and the Alliant FX/8. Even for these short vector the Alliant FX/8 vector gather/scatter hardware was fast enough to give the advantage to the column orientated vector algorithms of SLAP version 2.0. Undoubtedly, the cache system on the Alliant FX/8 played a major role.

Results for JPWH991						
	Cray Y/MP832		Alliant FX/8		Storage	
Method	ITER	Time	ITER	Time	Int	Real
SGECO		4.15		305.90	991	983,072
MA28		3.63		87.71	83,349	52,649
SSILUR	126	0.558	120	15.10	16,031	15,027
ilur	126	0.576	120	19.20		
SSDCGN	180	0.675	181	16.00	7,029	13,955
dscgn	180	0.591	181	28.10		
SSLUCN	31	0.350	30	7.80	16,031	18,991
ilucgn	31	0.309	30	9.74		
SSDBCG	40	0.151	37	3.39	7,029	13,955
dsbcg	40	0.133	37	5.92		
SSLUBC	16	0.194	14	4.01	16,031	18,991
ilubcg	16	0.173	14	5.18		
SSDCGS	23	0.045	27	2.59	7,029	13,955
SSLUCS	11	0.120	10	3.03	16,031	18,991
SSDOMN(7)	43	0.051	45	4.35	7,029	33,783
dsomn(7)	43	0.094	45	6.47		
SSLUOM(9)	14	0.099	14	2.88	16,031	50,715
iluomn(9)	14	0.090	14	3.78		
SSDGMR(9)	42	0.446	52	3.90	7,049	21,992
SSLUGM(11)	14	0.092	14	2.75	16,051	29,056

Table 6.6: Time and Storage results for JPWH991.

The JPWH991 problem also turned out to be negative definite and the results summarized in Table 6.6 reflect the solution of the recast system. Overall the nonsymmetric iterative methods were very competitive for this problem (even the conjugate gradient applied to the normal equations: `SSDCGN` and `SSLUCN`). The quickest (SLAP version 2.0) routine on the Cray Y/MP was `SSDCS`: over 80 times faster than `MA28` and over 92 times faster than `SGECO`. Again one can see the problems with short vectors on the Cray Y/MP, but not on the Alliant FX/8.

The SHERMAN2 problem was the hardest problem for the iterative methods: most of which could

Results for SHERMAN2						
	Cray Y/MP832		Alliant FX/8		Storage	
Method	ITER	Time	ITER	Time	Int	Real
SGECO		5.39		392.32	1,080	1,166,400
MA28		34.42		1,223.92	289,938	177,514
SSLUBC	14	0.417	14	9.85	50,521	53,748
ilubcg	14	0.685	14	23.48		
SSLUCS	8	0.327	8	8.01	50,521	53,748
SSLUOM(11)	15	0.327	15	8.70	50,521	85,079
iluomn(11)	19	0.495	15	17.48		
SSLUGM(11)	10	0.305	11	7.35	50,541	64,704

Table 6.7: Time and Storage results for SHERMAN2.

not solve the problem. The methods that did work worked quite well in comparison with the direct methods. The fastest routine was `SSLUGM(11)`, beating `MA28` and `SGECO`, on the Cray Y/MP, by a factor of 113 and 18, respectively. `MA28` suffered from a great deal of fill in and setting the pivoting parameter $u = 0.1$ had little effect.

Results for STEAM2						
	Cray Y/MP832		Alliant FX/8		Storage	
Method	ITER	Time	ITER	Time	Int	Real
SGECO		1.00		66.98	600	360,600
MA28		2.10		51.21	62,961	45,039
SSILUR	1	0.132	1	3.00	29,933	29,321
ilur	1	0.151	1	5.64		
SSDCGN	20	0.051	20	1.64	14,371	18,561
dscgn	20	0.155	20	6.97		
SSLUCN	1	0.140	1	3.14	29,933	31,721
ilucgn	1	0.169	1	6.35		
SSDBCG	8	0.020	8	0.75	14,371	18,561
dsbcg	8	0.063	8	3.03		
SSLUBC	1	0.138	1	3.23	29,933	31,721
ilubcg	1	0.164	1	6.28		
SSDCGS	5	0.008	5	0.54	14,371	18,561
SSLUCS	1	0.135	1	3.18	29,933	31,721
SSDOMN(7)	8	0.007	9	0.64	14,371	30,568
dsomn(7)	8	0.035	9	2.07		
SSLUOM(1)	1	0.132	1	2.93	29,933	32,922
iluomn(1)	1	0.151	1	5.55		
SSDGMR(5)	5	0.004	5	0.31	14,391	21,002
SSLUGM(5)	1	0.135	1	2.93	29,933	34,162

Table 6.8: Time and Storage results for STEAM2.

The easiest problem to solve in this set is the STEAM2 problem (Table 6.8). The iterative methods are vastly superior on this problem and give an indication of their power when utilized with appropriate preconditioners. This problem is block banded matrix (multiple diagonals made up of 4×4 blocks).

The diagonal block seems to be quite dominate. Hence, the incomplete *LU* preconditioner performance is spectacular and the diagonal scaling preconditioner is quite good. The best method is the `SSDGMR(5)` due to the fact that the incomplete factorization is not cheap to calculate. `SSDGMR(5)` is 525 times faster than `MA28` and 250 times faster than `SGECO` on the Cray Y/MP. The poor showing of `MA28` is due to a large amount of fill in. Reducing the pivoting parameter $u = 0.1$ does not change the fill in behavior.

7 Conclusions

The SLAP version 2.0 is a significant improvement over the version 1.0 code in terms robustness flexibility and speed. The latter improvement can be negated on the Cray X/MP and Y/MP class supercomputers when the number of non-zeros per column is quite small. The iterative methods presented here are quite competitive (even with very general preconditioners) with the direct methods on typical problems found in industry. Typical speed-up's of the iterative methods being between several to several hundred for the sample results. For the more difficult problems even more improvement would be obtained utilizing specialized preconditioners.

8 Acknowledgments

Many people have made contributions to SLAP 2.0, but the author would especially like to thank Anne Greenbaum for the initail development of the package and the conversion work she did to make the package more portable. The original code for the GMRES routines was obtained from Peter Brown and Alan Hindmarsh. Peter Brown participated in the effort to mold the routines into the SLAP scheme. Finally, NASA Ames NAS facility contributed non-trivial amounts of CPU time during the acceptance test of the Cray Y/MP to this project

References

[1] P. Concus, G. H. Golub, and D. P. O'Leary. *A Generalized Conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations.* In J. R. Bunch and D. J. Rose,

editors, **Sparse Matrix Computations**, pages 309–332, Academic Press, New York, 1976.

[2] I. S. Duff, R. Grimes, and J. Lewis. Sparse matrix test problems. *ACM-TOMS*, 0(0):To Appear, 1988.

[3] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. A. Watson, editor, *Numerical Analysis*, pages 73–89, Springer-Verlag, Letcure Notes in Mathematics 506, New York, 1976.

[4] K. W. Fong, T. H. Jefferson, and T. Suyehiro. *SLATEC Common Mathematical Library Source File Format.* Technical Report UCRL-53313, Lawrence Livermore National Laboratory, 1982.

[5] Anne Greenbaum. *Routines for Solving Large Sparse Linear Systems.* Tentacle News Letter, Livermore Computing Center, Lawrence Livermore National Laboratory, Livermore, CA 94550, January 1986.

[6] K. C. Jea and D. M. Young. *On the Simplification of Generalized Conjugate-Gradient Methods for Nonsymmetrizable Linear Systems.* **Linear Algebra Appl.**, 52/53:399–417, 1983.

[7] D. S. Kershaw. *The ICCG Method for the Iterative Solution of Systems of Linear Equations.* **J. Comp. Phys.**, 26:43–65, 1978.

[8] T. A. Manteuffel. *An Incomplete Factorization Technique for Positive Definite Linear Systems.* **Math. Comp.**, 34:473–497, 1980.

[9] Y. Saad and M. H. Schultz. *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems.* **SIAM J. Sci. Stat. Comput.**, 7(3):856–869, 1986.

[10] P. Sonneveld. *CGS, a fast Lanczos-type solver for the nonsymmetric linear systems.* Technical Report 84-16, Delft University of Technology, Department of Mathematics and Informatics, Julianalaan 132, 262B BL DELFT, 1984.